Concept · 19 mai 2026

← Précédent·Suivant →

Qu'est-ce que réellement SHA-256

Une empreinte mathématique qui tient en soixante-quatre caractères et qui change entièrement si une seule virgule du texte original est déplacée. Pourquoi nous l'appelons sceau de cire numérique.

Pour faire simple : Imaginez une machine qui lit n'importe quel texte et renvoie une séquence de 64 caractères. Si le texte est identique, la séquence est identique. Si vous déplacez une seule virgule, la séquence est complètement différente. Cette séquence est le cachet de cire numérique.

L'idée simple derrière le nom technique

Imaginez qu'il existe une machine avec une seule fente et un seul écran. Par la fente, vous introduisez un texte : un mot, une phrase, un roman entier. Sur l'écran apparaît, quelques instants plus tard, une séquence d'exactement soixante-quatre caractères. Cette séquence, pour le lecteur professionnel, nous l'appelons *hash* ou *résumé cryptographique* ; pour le lecteur général, nous pouvons l'appeler pour l'instant une empreinte mathématique du texte, tout comme l'empreinte digitale l'est pour une personne.

Si vous introduisez le même texte deux fois, la machine affiche la même empreinte les deux fois. Si vous introduisez un texte légèrement différent — une seule virgule déplacée, une majuscule qui devient minuscule — la machine affiche une empreinte complètement différente de la première. Pas semblable : différente. Ces deux propriétés ensemble — le déterminisme et la sensibilité — sont l'idée simple. Tout le reste du SHA-256 est la machinerie qui les fait bien respecter.

Il convient de dire dès le départ ce que la machine ne fait pas. Elle ne chiffre pas le texte. Elle ne le cache pas. Elle ne le garde pas. La machine regarde le texte, calcule l'empreinte, et oublie le texte. L'empreinte ne permet pas de reconstruire le texte qui l'a produite ; elle permet seulement, étant donné un texte candidat, de vérifier s'il coïncide ou non avec l'original. C'est pourquoi nous disons que c'est un résumé *à sens unique* : on y va, on n'en revient pas.

Un hash n'est pas la même chose que chiffrer

La confusion est fréquente et il convient de la dissiper : chiffrer et hacher sont des opérations distinctes. Chiffrer consiste à transformer un texte de manière à ce que seul le détenteur de la clé puisse le ramener à sa forme originale. Hacher consiste à produire une empreinte du texte à partir de laquelle le texte original ne peut jamais être récupéré, avec ou sans clé. La première est réversible par conception ; la seconde, irréversible par conception.

La conséquence pratique importe. Lorsqu'une application dit « nous gardons votre mot de passe chiffré », il y a quelqu'un qui possède la clé pour le déchiffrer — l'application elle-même, en tout cas. Lorsqu'une application dit « nous gardons votre mot de passe haché », l'application elle-même ne peut pas lire le mot de passe original même si elle le voulait ; elle peut seulement vérifier si celui que vous tapez produit à nouveau la même empreinte. Le second modèle, bien fait, est bien préférable au premier pour stocker des mots de passe. Nous verrons plus tard pourquoi « bien fait » exige un peu plus que le seul SHA-256.

Les quatre propriétés qui rendent un hash cryptographique utile

Une fonction de hachage qui mérite l'adjectif *cryptographique* remplit quatre propriétés :

1. **Déterminisme.** La même entrée produit toujours la même empreinte.
2. **Effet avalanche.** Un petit changement dans l'entrée produit une empreinte complètement différente, sans ressemblance visible avec la précédente.

3. **Résistance à l'inversion.** Étant donné une empreinte, il n'est pas informatiquement viable de trouver le texte qui l'a produite.
4. **Résistance aux collisions.** Il n'est pas informatiquement viable de trouver deux textes différents qui produisent la même empreinte.

« Pas informatiquement viable » ne signifie pas « mathématiquement impossible ». Cela signifie que le coût en temps, en énergie et en argent pour y parvenir dépasse d'ordres de grandeur la somme de toute la capacité de calcul raisonnablement disponible. Pour le SHA-256, cette limite se mesure en milliers de milliards d'années, même pour les approches les plus optimistes avec un matériel spécialisé. Ce qui, pour les besoins pratiques du lecteur, revient au même que « c'est impossible ».

SHA-256, en particulier

Le nom dit tout. SHA est l'acronyme de *Secure Hash Algorithm* : algorithme de hachage sécurisé. Le nombre 256 indique la taille de l'empreinte en bits : deux cent cinquante-six bits, soit trente-deux octets, qui, affichés en hexadécimal, correspondent aux soixante-quatre caractères que le lecteur reconnaît déjà. Le standard a été publié par le NIST américain, l'organisme qui normalise ce type de fonctions, en 2001 dans le cadre de la famille SHA-2 ; la version actuelle du standard, FIPS 180-4, date de 2015.

Pour ceux qui ne savent pas encore ce que sont les bits et les octets :

```
1 bit     →  0 ou 1       (un interrupteur : allumé ou éteint)
1 octet   →  8 bits       (256 combinaisons possibles)
32 octets →  256 bits     (l'empreinte SHA-256)
```

Le nombre 256 à la fin du nom indique la taille de l'empreinte en bits. En hexadécimal — un système de numération à seize symboles au lieu de dix — ces 256 bits tiennent en exactement 64 caractères. Ce sont les 64 caractères que vous voyez au bas de chaque Cuaderno.

Les dimensions méritent un instant. Deux cent cinquante-six bits permettent deux à la puissance deux cent cinquante-six valeurs différentes : un nombre à soixante-dix-huit chiffres décimaux, plusieurs ordres de grandeur supérieur au nombre estimé d'atomes dans l'univers observable. Chaque texte au monde — chaque livre, chaque e-mail, chaque message — tombe sur l'une de ces valeurs. La probabilité que deux textes différents coïncident par hasard est, à toutes fins pratiques, indiscernable de zéro.

À quoi cela ressemble en code

En Zig, le langage dans lequel nous écrivons les pièces qui soutiennent Solo2, calculer le sceau SHA-256 d'un texte ressemble à ceci :

```
const std = @import("std");

const texto = "Cuadernos Lacre";
var resumen: [32]u8 = undefined;
std.crypto.hash.sha2.Sha256.hash(texto, &resumen, .{});
```

Nous venons de demander à la bibliothèque standard de Zig de calculer le SHA-256 du texte entre guillemets. Après l'appel, la variable *resumen* contient les trente-deux octets qui composent le sceau dans sa forme brute ; lorsqu'ils sont affichés à l'écran en hexadécimal, ce sont les soixante-quatre caractères qui apparaissent au bas de cet article. Si nous changions *Cuadernos Lacre* en *Cuadernos lacre* — une majuscule en moins — le sceau changerait entièrement. C'est, en cinq lignes, la propriété centrale qui soutient le reste. Pour ceux qui veulent voir comment cela fonctionne en interne, nous incluons à la fin de l'article une version lisible de l'algorithme avec des commentaires étape par étape.

Pourquoi nous l'appelons sceau de cire

Dans la correspondance européenne du XVe au XIXe siècle, la cire fermait la lettre. Une goutte de cire fondue, un sceau pressé dessus, et la lettre était marquée de manière unique. Elle ne protégeait pas le contenu du curieux décidé — le papier pouvait être lu à contre-jour, la cire pouvait être brisée — mais elle le prouvait. Toute altération de la fermeture était visible par le destinataire avant même d'ouvrir le papier. La cire n'empêchait pas le dommage ; elle le déclarait.

Le SHA-256 du corps de chaque Cuaderno remplit la même fonction dans sa version numérique. Si un seul mot de l'article changeait entre le moment où il a été publié et le moment où vous le lisez, le sceau hexadécimal au bas du texte ne correspondrait plus au SHA-256 du texte que vous avez devant vous. N'importe quel lecteur avec cinq lignes de code pourrait le vérifier. La publication ne peut pas réécrire son histoire sans que le sceau ne la trahisse. Elle ne protège pas contre le dommage ; elle le rend vérifiable.

Ce qu'un hash n'est pas

Quatre usages sont parfois demandés au SHA-256 alors qu'ils ne lui correspondent pas :

1. **Chiffrer.** Un hash résume ; il ne cache pas. Si vous voulez que le texte ne puisse pas être lu, vous devez le chiffrer, pas le hacher.
2. **Authentifier l'auteur.** Un hash ne dit pas qui a écrit le texte, seulement quel texte a été haché. Pour associer une paternité, il faut une signature cryptographique sur le hash, pas le hash seul.
3. **Stocker des mots de passe.** Il y a ici un piège qu'il convient de comprendre. Le SHA-256 est conçu pour être très rapide — ce qui est bon pour beaucoup de choses, mais mauvais pour celle-ci. Un attaquant doté d'un matériel spécialisé peut tester des milliards de mots de passe par seconde contre un hash SHA-256 jusqu'à trouver le vôtre. Pour enregistrer des mots de passe, il faut utiliser des fonctions de dérivation de clé délibérément lentes comme Argon2, scrypt ou bcrypt, combinées à un *sel* (une donnée aléatoire unique par utilisateur, qui empêche deux personnes ayant le même mot de passe d'avoir le même hash).
4. **Lire le hash comme identifiant de l'auteur.** Ce n'est pas le cas. Un hash identifie le contenu. Si deux personnes hachent le mot *bonjour* avec SHA-256, elles obtiennent toutes les deux le même résumé — et c'est la propriété centrale, pas un défaut : s'il s'agissait de résumés différents, nous ne pourrions pas vérifier la coïncidence entre ce qui est publié et ce qui est reçu.

Où apparaît SHA-256 dans votre quotidien

Même si vous ne le voyez pas, le SHA-256 soutient une bonne partie de ce que vous utilisez quotidiennement sur Internet. La chaîne de blocs Bitcoin se construit en enchaînant le SHA-256 de chaque bloc au suivant ; modifier un bloc passé oblige à recalculer toute la chaîne ultérieure. Git, le système avec lequel est versionné le code de la moitié du monde, identifie chaque commit par le SHA-256 (dans les versions récentes) ou par son prédécesseur SHA-1 (dans les versions plus anciennes) de son contenu complet. Les certificats HTTPS qui vérifient l'identité d'un site web lorsque vous y entrez portent une empreinte SHA-256 associée. Les téléchargements de logiciels sont souvent accompagnés d'un SHA-256 publié par le développeur pour que vous puissiez vérifier que le fichier n'a pas été altéré en cours de route. Et, comme nous l'avons dit, au bas de chaque Cuadernos Lacre.

Pour le lecteur professionnel

Quatre rappels opérationnels pour celui qui décide ou audite des systèmes :

1. Hash n'est pas chiffrement. Si un fournisseur confond les deux termes dans sa documentation technique, il convient de demander ce qu'il veut dire exactement.
2. Pour stocker des mots de passe, il ne faut jamais utiliser SHA-256 seul. SHA-256 est too rapide pour cette tâche (voir point 3 de *Ce qu'un hash n'est pas*). Le standard actuel est **Argon2id** : lent par conception, configurable selon la capacité du serveur, combiné à un *sel* aléatoire différent par utilisateur.
3. Pour l'intégrité des documents — contrats, dossiers, fichiers — SHA-256 reste la norme de référence. C'est celui utilisé par les horodateurs qualifiés dans l'UE.
4. Pour une conservation à long terme (décennies), il convient de calculer et d'archiver également un SHA-3 ou un SHA-512 aux côtés du SHA-256 ; la prudence cryptographique recommande de ne pas s'appuyer sur une seule fonction lors d'archivages centenaires.

Techniquement, cette structure itérée — où l'état intermédiaire est conservé entre les blocs d'entrée — est connue sous le nom de construction de **Merkle-Damgård**, le modèle sur lequel reposent SHA-1, SHA-2 (y compris SHA-256) et bien d'autres fonctions de hachage classiques. SHA-3, en revanche, abandonne Merkle-Damgård au profit d'une architecture différente appelée *éponge*.

Comment fonctionne SHA-256, étape par étape, en termes simples

Imaginez que vous ayez monté le circuit de dominos le plus élaboré au monde : des milliers de pièces, des dizaines de bifurcations, des ponts mécaniques et des rampes traversant toute la pièce, soigneusement placés pièce par pièce.

Si vous touchez la première pièce, la chaîne tombe selon une séquence précise et reproductible. Même montage, même touche initiale → motif final identique de pièces tombées, encore et encore.

Voici ce qui est intéressant : déplacez **une seule pièce** d'un demi-centimètre sur le côté avant de commencer et touchez à nouveau. Une rampe qui devait s'activer reste inerte, un pont ne tombe pas, une bifurcation différente se déclenche. Le motif final de pièces au sol est complètement méconnaissable par rapport au premier.

SHA-256 est mathématiquement ce circuit. Le texte que vous écrivez est la position initiale des pièces. L'algorithme est la touche qui libère la cascade. Et le résultat final — ce que nous appelons *hash* — est la photo figée du sol quand tout s'est arrêté. Changez une seule virgule du texte original et la photo sera radicalement différente. C'est aussi simple, et aussi drastique que cela.

Étape 1. Traduire le texte en pièces binaires. Les ordinateurs ne comprennent pas les lettres ; ils les traduisent d'abord en nombres (ASCII) et les nombres en binaire (des uns et des zéros). Chaque lettre devient 8 pièces blanches ou noires : le *A* est `01000001`, le *B* est `01000010`, l'espace est `00100000`. Votre texte entier — un mot, un contrat, un roman — devient une longue file de pièces blanches et noires.

Étape 2. Remplir jusqu'à la taille standard. Le circuit traite la file par *tronçons* de 512 pièces exactement. Si votre message n'atteint pas un multiple de 512, on ajoute une pièce marqueuse (de valeur `10000000`) juste après le texte, puis des zéros jusqu'à compléter le tronçon. Les 64 dernières positions de chaque tronçon sont réservées pour noter la longueur originale du texte. Ainsi, le circuit sait toujours où s'est arrêté le contenu réel et où a commencé le remplissage.

Étape 3. Placer les huit pièces maîtresses. Avant de commencer, nous plaçons sur la table **huit pièces maîtresses** dans une position initiale précise. Ces huit pièces ne sont pas un secret : leur valeur initiale est fixée par une règle mathématique publique (les racines carrées des huit premiers nombres premiers — 2, 3, 5, 7, 11, 13, 17, 19 — et les premiers bits de la partie décimale de chaque racine). Tout le monde, aux quatre coins de la planète, commence avec les mêmes huit pièces maîtresses dans la même position. Leur destin est d'être poussées et transformées par l'avalanche.

Étape 4. La grande avalanche : soixante-quatre rounds de poussées. C'est ici que le spectacle commence. Le premier tronçon de 512 pièces de votre texte vient percuter les huit pièces maîtresses. Mais elles ne tombent pas d'un coup : le mécanisme exécute **soixante-quatre rounds consécutifs**. À chaque round, il effectue trois opérations avec les pièces :

- **Le Manège** (rotation). Les pièces bougent en cercle : celles de droite passent à gauche. Aucune pièce n'est perdue ni ajoutée ; elles sont simplement réordonnées en faisant un tour complet de manège. C'est une manière économique et réversible de redistribuer l'information.
- **L'Entonnoir Logique** (XOR). Les pièces passent par un entonnoir qui les compare deux à deux : si les deux sont de la même couleur, une blanche sort ; si elles sont différentes, une noire sort. C'est l'opération la plus simple de la logique binaire, mais combinée aux rotations du manège, elle devient extrêmement puissante pour mélanger l'information sans la perdre.
- **Le Débordement** (somme modulaire). Le résultat est additionné à une *pièce de poussée constante* tirée d'une liste publique de soixante-quatre constantes (les racines cubiques des soixante-quatre premiers nombres premiers). Si la somme génère des pièces supplémentaires qui ne tiennent pas dans l'espace de 32 pièces prévu, ces pièces excédentaires sont écartées. La table n'a de place que pour 32 pièces, pas une de plus.

À la fin du round soixante-quatre, chacune des pièces du tronçon de votre texte a influencé la position des huit pièces maîtresses. L'énergie de la poussée a voyagé dans tout le circuit.

Étape 5. Ajouter le tronçon suivant (sans réinitialiser). Si votre texte était long et qu'il reste un autre tronçon de 512 pièces à traiter, **le circuit ne se réinitialise pas**. Les huit pièces maîtresses restent telles que la première avalanche les a laissées, et le second tronçon est lancé contre elles pour activer soixante-quatre autres rounds. C'est comme ajouter une nouvelle pièce pleine de dominos à la suite de celle qui vient de tomber : le désordre de la première conditionne entièrement la façon dont la seconde tombera.

Étape 6. Faire la photo finale. Quand il ne reste plus de tronçons à traiter, l'avalanche s'arrête. Nous regardons la position finale dans laquelle se trouvent les huit pièces maîtresses. Nous traduisons leur configuration en un code de lettres et de chiffres en système hexadécimal. Le résultat est une chaîne d'exactement soixante-quatre caractères : c'est votre sceau SHA-256.

Quatre propriétés découlent d'elles-mêmes de la façon dont le circuit est monté :

1. **Déterminisme.** Le même texte produit toujours la même photo finale, sur n'importe quel ordinateur au monde. Zéro aléatoire, zéro surprise.
2. **Effet avalanche.** Une virgule ajoutée, une majuscule changée, un accent oublié : la photo devient complètement méconnaissable. C'est la sensibilité extrême que nous avons déjà décrite au début.
3. **Une seule direction.** À partir de la photo finale, vous ne pouvez pas reconstruire le texte original. Les rotations, les entonnoirs et les débordements détruisent toute l'information directionnelle sur *l'origine de chaque bit* et ne conservent que *ce qui a été ajouté au total*.
4. **Résistance aux collisions.** En vingt-cinq ans de cryptanalyse publique, personne n'a réussi à trouver deux textes différents dont les photos finales coïncident. Et la difficulté d'y parvenir dépasse les capacités de calcul de toute civilisation raisonnablement imaginable.

L'appendice de code qui suit implémente exactement ces six étapes en Zig. Vous pouvez maintenant le lire en sachant ce que signifie chaque opération sur les bits, au lieu d'accepter les manipulations aveuglément.

Glossaire technique

Pour le lecteur qui souhaite comprendre ce que fait chaque opération. Passez-le librement : l'article reste compréhensible sans lui.

ASCII et Unicode — comment les lettres deviennent des nombres. Les ordinateurs ne voient pas de lettres ; ils voient des nombres. Un standard appelé **ASCII** (*American Standard Code for Information Interchange*, de 1963) assigne à chaque caractère du clavier un nombre spécifique : le *A* est 65, le *B* est 66, le *a* est 97, le *0* est 48, l'espace est 32, la virgule est 44. Les systèmes modernes l'étendent avec **Unicode**, qui assigne un nombre à chaque caractère de chaque alphabet du monde : cyrillique, arabe, chinois, japonais, et même les emojis. Quand vous tapez un caractère ou ouvrez un fichier texte, l'ordinateur lit le nombre sous-jacent, pas la forme à l'écran. SHA-256 travaille sur ces nombres, traitant tout texte comme une longue séquence de chiffres. C'est pourquoi il peut sceller un article en espagnol, un poème en japonais et un fichier binaire avec le même algorithme.
XOR — le comparateur bit à bit. XOR (prononcé *« exor »*, de l'anglais *exclusive or*, « ou exclusif ») est l'une des opérations les plus simples qu'un ordinateur puisse effectuer avec deux nombres binaires. Il compare deux bits position par position et renvoie : **1** si exactement l'un des deux est 1 (l'un mais pas les deux), **0** si les deux sont identiques (les deux à 0 ou les deux à 1). Exemple : le XOR de `1010` et `1100` est `0110`. Il possède une propriété remarquable : il est réversible — si vous faites un XOR deux fois avec la même clé, vous revenez à l'original. C'est pourquoi c'est le cheval de bataille de la cryptographie : il mélange les bits sans perdre d'information, mais le résultat ne révèle rien sur les entrées si vous n'en connaissez pas l'une d'elles.
Hexadécimal — compter en base 16. Presque tous les nombres du quotidien utilisent dix chiffres (0-9). L'hexadécimal en utilise seize : les habituels 0-9 plus six lettres qui représentent les valeurs suivantes : `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15. Pourquoi seize ? Parce que les ordinateurs réfléchissent par groupes de quatre bits, et quatre bits peuvent représenter exactement seize valeurs différentes — ainsi, un caractère hexadécimal correspond proprement à quatre bits. Une empreinte SHA-256 mesure 256 bits, soit exactement **64 caractères hexadécimaux**. Si nous l'écrivions en décimal courant, elle occuperait environ 78 chiffres et serait plus encombrante. Le choix est esthétique et compact ; le nombre de fond est le même.
Rotation de bits — le manège binaire. Imaginez une file de sept ampoules, certaines allumées (1) et d'autres éteintes (0) : `1 0 1 1 0 0 1`. Faire pivoter vers la droite d'une position consiste à prendre l'ampoule tout à droite, à l'amener à l'extrémité gauche et à décaler les autres d'une place vers la droite : `1 1 0 1 1 0 0`. Aucune ampoule n'est perdue ni ajoutée : elles dansent simplement en cercle. SHA-256 utilise la rotation de bits des centaines de fois dans chaque calcul ; c'est une manière économique et sans perte de redistribuer l'information au sein de l'état.
Constantes *« nothing-up-my-sleeve »* — pourquoi elles proviennent de nombres premiers. Les huit pièces maîtresses et les soixante-quatre constantes de round de SHA-256 n'ont pas été choisies au hasard. Elles proviennent des racines carrées et cubiques des premiers nombres premiers. Pourquoi ? Parce que leurs concepteurs voulaient des constantes *« sans rien dans les manches »* (*« nothing-up-my-sleeve »*) : des valeurs dont n'importe qui peut vérifier l'origine. Si quelqu'un vous disait *« faites-moi confiance : utilisez ce nombre aléatoire de 32 bits »*, vous suspecteriez à juste titre une faiblesse cachée ou une porte dérobée. Mais n'importe qui muni d'une calculatrice peut vérifier que les 32 premiers bits de la racine

carrée de 2 sont `0x6a09e667`. Les valeurs sont mathématiques, publiques et reproductibles : aucun piège caché ne peut se glisser dans la recette.

Appendice : SHA-256 en code lisible

Cet appendice s'adresse au lecteur qui souhaite voir l'algorithme de l'intérieur. Il s'agit d'une implémentation didactique en Zig qui suit la spécification FIPS 180-4. Ce n'est pas la version utilisée par Solo2 — la vraie se trouve dans `std.crypto.hash.sha2.Sha256` de la bibliothèque standard de Zig, optimisée et auditée. Mais l'algorithme est le même : ce que vous voyez ici est, étape par étape, ce qui se passe lorsque cet appel de cinq caractères effectue son travail.

```
const std = @import("std");

// SHA-256 — implementación didáctica.
// Sigue la especificación FIPS 180-4. Prioriza la claridad sobre la
// velocidad y la robustez frente a entradas hostiles. Para producción,
// usa std.crypto.hash.sha2.Sha256, que está optimizada y auditada.

// H0: las ocho palabras del estado inicial. Primeros 32 bits de la parte
// fraccionaria de las raíces cuadradas de los primeros ocho primos
// (2, 3, 5, 7, 11, 13, 17, 19).
const H0 = [_]u32{
    0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
    0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19,
};

// K: 64 constantes de ronda. Primeros 32 bits de la parte fraccionaria
// de las raíces cúbicas de los primeros 64 primos.
const K = [_]u32{
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2,
};

// Rotación circular a la derecha de un u32.
inline fn rotr(x: u32, n: u5) u32 {
    return std.math.rotr(u32, x, n);
}

// Lee 4 bytes consecutivos como un u32 big-endian.
inline fn readU32(b: []const u8) u32 {
    return @as(u32, b[0]) << 24 | @as(u32, b[1]) << 16 | @as(u32, b[2]) << 8 | @as(u32, b[3]);
}

// Escribe un u32 como 4 bytes consecutivos big-endian.
inline fn writeU32(b: []u8, v: u32) void {
    b[0] = @truncate(v >> 24);
    b[1] = @truncate(v >> 16);
    b[2] = @truncate(v >> 8);
    b[3] = @truncate(v);
}

// Compresión de un bloque de 64 bytes sobre el estado del hash. Sigue §6.2.2 de FIPS 180-4.
fn compress(state: *[8]u32, block: [16]u32) void {

    // 1. Expansión del schedule: 16 palabras → 64. Las nuevas se obtienen
    //    combinando cuatro anteriores con dos funciones de mezcla (s0 y s1)
    //    que usan rotación, XOR y desplazamiento. El "+%" es suma con
    //    truncado u32 (overflow-wrap), tal como exige el estándar.
    var w: [64]u32 = undefined;
    for (0..16) |i| w[i] = block[i];
    for (16..64) |i| {
        const s0 = rotr(w[i-15], 7) ^ rotr(w[i-15], 18) ^ (w[i-15] >> 3);
        const s1 = rotr(w[i-2], 17) ^ rotr(w[i-2], 19) ^ (w[i-2] >> 10);
```

```
        w[i] = w[i-16] +% s0 +% w[i-7] +% s1;
    }

    // 2. Variables de trabajo: copia del estado actual.
    var a = state[0]; var b = state[1]; var c = state[2]; var d = state[3];
    var e = state[4]; var f = state[5]; var g = state[6]; var h = state[7];

    // 3. 64 rondas de mezcla no lineal.
    //    S1, S0  : combinaciones rotacionales de 'e' y 'a'.
    //    ch      : "choose" — multiplexor bit a bit, elige entre f y g según e.
    //    maj     : "majority" — bit mayoritario entre a, b, c.
    //    t1 + t2 : se inyecta al top de la cascada cada ronda.
    for (0..64) |i| {
        const S1 = rotr(e, 6) ^ rotr(e, 11) ^ rotr(e, 25);
        const ch = (e & f) ^ (~e & g);
        const t1 = h +% S1 +% ch +% K[i] +% w[i];
        const S0 = rotr(a, 2) ^ rotr(a, 13) ^ rotr(a, 22);
        const maj = (a & b) ^ (a & c) ^ (b & c);
        const t2 = S0 +% maj;
        h = g; g = f; f = e; e = d +% t1;
        d = c; c = b; b = a; a = t1 +% t2;
    }

    // 4. Acumular las variables de trabajo en el estado.
    state[0] +%= a; state[1] +%= b; state[2] +%= c; state[3] +%= d;
    state[4] +%= e; state[5] +%= f; state[6] +%= g; state[7] +%= h;
}

// Hash completo: procesa el mensaje en bloques, padea el último, escribe el resumen.
pub fn sha256(msg: []const u8, out: *[32]u8) void {
    var state = H0;
    var block: [64]u8 = undefined;
    var block_w: [16]u32 = undefined;

    // Procesar bloques completos del mensaje original.
    var i: usize = 0;
    while (i + 64 <= msg.len) : (i += 64) {
        @memcpy(block[0..64], msg[i..i+64]);
        for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4]);
        compress(&state, block_w);
    }

    // Padding del último bloque: byte 0x80, después ceros, después la
    // longitud original (en bits) como u64 big-endian en los 8 últimos bytes.
    const remaining = msg.len - i;
    @memcpy(block[0..remaining], msg[i..]);
    block[remaining] = 0x80;
    const bit_len: u64 = @as(u64, msg.len) * 8;

    if (remaining + 1 + 8 <= 64) {
        // El padding cabe en el mismo bloque.
        for (remaining + 1..56) |k| block[k] = 0;
        var k: usize = 0;
        while (k < 8) : (k += 1) block[56 + k] = @truncate(bit_len >> @as(u6, @intCast((7 - k) * 8)));
        for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4]);
        compress(&state, block_w);
    } else {
        // El padding requiere un bloque adicional.
        for (remaining + 1..64) |k| block[k] = 0;
        for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4]);
        compress(&state, block_w);
        for (0..56) |k| block[k] = 0;
        var k: usize = 0;
        while (k < 8) : (k += 1) block[56 + k] = @truncate(bit_len >> @as(u6, @intCast((7 - k) * 8)));
        for (0..16) |j| block_w[j] = readU32(block[j*4..j*4+4]);
        compress(&state, block_w);
    }

    // Escribir el estado final como 32 bytes big-endian.
    for (0..8) |j| writeU32(out[j*4..j*4+4], state[j]);
}
```

```
// Ejemplo de uso.
pub fn main() void {
    var resumen: [32]u8 = undefined;
    sha256("Cuadernos Lacre", &resumen);
    for (resumen) |byte| std.debug.print("{x:0>2}", .{byte});
    std.debug.print("\n", .{});
    // Imprime: ae6bdea6bbf5476889e0651a31f3dc1612fc61497477e21a95cabae2a6886c3e
}
```

Toute réécriture dans un autre langage qui suit la même structure — constantes initiales, expansion du schedule, soixante-quatre tours, accumulation — produit le même résultat. L'algorithme n'a pas de secrets : sa valeur réside dans le fait que les propriétés énumérées plus haut continuent d'être valables après deux décennies de cryptanalyse publique par des milliers d'yeux.

Si vous revenez au bas de cet article, vous verrez un sceau hexadécimal de soixante-quatre caractères. C'est le SHA-256 du texte que vous venez de lire, dans cette langue. Si nous traduisions l'article, le sceau serait différent ; si un mot de la version espagnole changeait, le sceau espagnol changerait. Le sceau ne protège pas le contenu — d'autres outils sont là pour cela — mais il l'identifie de manière unique. Et cela, aussi modeste que cela puisse paraître, suffit pour qu'aucune étape de la chaîne éditoriale ne puisse altérer ce qui a été dit sans que cela ne se remarque. Le reste — chiffrer, signer, identifier — se construit sur cette idée simple.

Note éditoriale : quand ces Cuadernos nomment des entreprises ou des produits, ce n'est pas pour accuser. Ceux qui les construisent font un travail que des millions de personnes utilisent et apprécient. Ce que nous soulignons est structurel — le modèle, pas la marque. Les marques apparaissent comme exemples car ce sont celles que le lecteur reconnaît.

Sources et lectures complémentaires

- NIST — *FIPS PUB 180-4: Secure Hash Standard (SHS)*, août 2015. Spécification officielle de la famille SHA-2, y compris SHA-256.
- RFC 6234 — *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, IETF, mai 2011. Version normative pour les implémenteurs.
- Ferguson, N.; Schneier, B.; Kohno, T. — *Cryptography Engineering: Design Principles and Practical Applications* (Wiley, 2010). Les chapitres 5 et 6 traitent des fonctions de hachage et de leurs utilisations légitimes et illégitimes.
- Nakamoto, S. — Bitcoin: A Peer-to-Peer Electronic Cash System (2008). Exemple pratique de l'utilisation de SHA-256 pour enchaîner les blocs dans une structure immuable par construction.
- Règlement (UE) 910/2014 (eIDAS) — cadre pour les prestataires de services d'horodatage électronique qualifiés. SHA-256 est la fonction de référence pour les signatures et sceaux électroniques qualifiés émis dans l'UE.
- Implémentation de référence en Zig : `std.crypto.hash.sha2.Sha256` dans le dépôt officiel du langage (*github.com/ziglang/zig* → `lib/std/crypto/sha2.zig`). C'est la version optimisée et auditée qu'utilise Solo2. Utile pour comparer avec l'implémentation didactique de l'appendice.

← PrécédentSchrems II, cinq ans aprèsSuivant →Kill switch et capture institutionnelle

Lectures recentes

- Analyse · 18 mai 2026 Confidentialité réelle vs apparente : les questions à se poser
- Analyse · 18 mai 2026 Self-hosting comme pratique professionnelle
- Concept · 18 mai 2026 Les 24 mots : qu'est-ce qu'une identité cryptographique

Emportez cet article où vous en avez besoin.

↓ Markdown ↓ Texte brut ↓ PDF

Le fichier est téléchargé sur votre appareil. À partir de là, vous pouvez l'enregistrer, l'importer dans Solo2 ou le partager comme vous le souhaitez. Cuadernos ne décide pas de la destination pour vous.

Cachet de cire · SHA-256 d3da6b6f510911a5d68308ea477d1bd9a927b2091e1cff70e2cbb505fb48856a

Cuadernos Lacre · Une publication de Menzuri Gestión S.L. ·
écrite par R.Eugenio · éditée par l'équipe de Solo2.